# An approach for processing and document flow automation for Microsoft Word and LibreOffice Writer file formats

Pavlo V. Zahorodko[1], Pavlo V. Merzlykin[1]

[1]*Kryvyi Rih State Pedagogical University, 54 Gagarin Ave., Kryvyi Rih, 50086, Ukraine*

### Abstract

The rapid growth of modern information technologies influences all aspects of human life. Companies all over the world are adopting new approaches to solve business problems, such as diverse automation, by using information technologies. Automation substitutes routine human work and noticeably increases efficiency. This research examines different approaches to document automation. Basic concepts of document processing using XML and existing solutions have been reviewed and a library based on LibreOffice UNO API has been designed and implemented. The library contains different helpers, wrappers, and processing tools to create an additional layer of abstraction. Moreover, the library is aimed at simplifying processing, working, and converting documents, which might considerably optimize a process of creating document reports generators.

### Keywords

document processing, automation, library, OpenDocument, Office Open XML

## 1. Introduction

A significant amount of organizations, companies, and educational institutions deal frequently with different document-related processes. Eventually, the growth of a company causes a demand on optimizing processes. Documentation generators are one of the earliest and substantial stages of business processes automation [1].

According to McKinsey Global Institute [2], which is a part of the worldwide management-consulting firm McKinsey&Company, from 9 to 26 percent of working hours could be saved by automation. Additionally, with a midpoint of 15 percent, about 30 percent of working places could be displaced by 2030, which is equivalent to 400 million full-time working days. In addition, the research admits that about 50 percent of working time, which is spent on different types of work, might be optimized with automation.

Hospitals, as well as other organizations, work with an immense amount of documents. According to Steve Wilson's paper on Electronic Health Reporter website [3], every day doctors have to deal with a large amount of different documents, starting from physician agreements

*CS&SE@SW 2021: 4th Workshop for Young Scientists in Computer Science & Software Engineering, December 18, 2021, Kryvyi Rih, Ukraine*

mongolzzz21@gmail.com (P. V. Zahorodko); ipmcourses@gmail.com (P. V. Merzlykin)
https://kdpu.edu.ua/personal/pvmerzlykin.html (P. V. Merzlykin)
0000-0002-4017-7172 (P. V. Merzlykin)

and credential documents to time sheets and other organizational forms. Undoubtedly, it is hard to handle or search through such a number of paper documents in comparison with digital ones. Another, surely important, reason to use automation is working with patients. Digital forms help to avoid human interaction, which has become crucial due to the COVID-19 pandemic. In addition, digital forms might help simplify the whole process of requesting prescriptions.

Whilst the described problem seems completely explored, it is not exactly so. Many existing implementations are proprietary, that is to say you could not obtain their source code easily. This leads to the fact that it is hardly possible to launch software locally for your company or set it up preferably, for instance, choose a web-server or database. Moreover, the assortment of the supported documents is usually meager and often includes only Microsoft Office formats. Another hot topic is privacy. If processed documents contain users' sensitive or corporate data, you could not trust proprietary cloud services you are not able to control. Moreover, it could be simply considered illegal in some countries to transfer personal data to 3rd parties servers. Consequently, it is critical for document automation systems to allow users to have control over their data. The aforesaid leads us to the reasons why we decided to develop our own document management system as an attempt to solve the mentioned problems.

## 2. Overview

A review of scientific literature [4, 5, 6, 7] on the topic of document flow automation showed that the topic is relevant. But due to the lack of access to the source code, we will examine only those implementations that are open or provide, at least partially, free trial access to the service.

Let's take a look at the proprietary document processing systems.

Hypatos [8, 9] is a workflow automation system which uses artificial intelligence, namely Cognitive Process Automation (CPA) technology. It is a fairly high-quality and professional tool. It supports AWS and cloud storage. Both API and free version are available.

DocuPhase [10] is a system for automating business processes. It supports web forms that allow one to generate ready-made PDFs. It also features a document management system with user-friendly interface for processing and managing files shared among different departments.

Docupilot [11] is an automation and documentation generation system. It supports working with cloud services such as Zapier, DropBox, Docusign. It has a good templating engine with conditional statements, tables and loops support. It could handle docx, pptx, pdf or a custom, created with a WYSIWYG editor, template. It also supports email messages sending. There is documentation and examples of using the internal API.

Contactbook [12] is a platform for organizing, storing and processing documents. The service supports docx and pdf files. An integration with 3000+ programs has been implemented. A public API as available as well.

Now let's take a look at the open-source applications.

One of these is Docassemble [13], an open-source system for working with web forms and documents. The system is implemented with Python, YAML and Markdown. It is focused on "Interview" questions. That is to say, one web form is divided into several questions and at the end you can get a result. It supports YAML code in configuration files. With Markdown, one could dynamically create PDF, RTF, and DOCX files.

M2Doc [14] is an open-source plug-in for automating MS Word files processing. There are add-ons for MS Word and Eclipse IDE. The generator takes input data from a generator configuration `.genconf`. One is able to work with the original Java API.

Summarizing this section, the reviewed systems are competitive and powerful tools. But, they have the following disadvantages:

1. Static patterns. Most tools use only one proposed pattern for fields filling. It means that only system prefix and suffix ought be used in templates. For instance, with the prefix `{{` and the suffix `}}`, field definition would look like {{field}}.
2. Solely Microsoft Word formats support. Most mentioned systems don't support LibreOffice file format or other similar formats. However Microsoft products usage is not always possible or acceptable by some companies.
3. No internal converters. Sometimes it is needed to convert a document into different format than docx or pdf.

Thus, it was decided to design our own system for documents processing that would satisfy our needs.

## 3. Approaches in document processing

Document management system needs a core document processing tool. There are a few different approaches in Microsoft Office and LibreOffice documents processing. We will overview the most popular: XML processing and frameworks.

Microsoft Office and LibreOffice documents are basically archives with all content inside. Most of the files inside are XML files. They represent document's structures, styles, metadata, settings, and other configurations.

Microsoft Office documents (doc and docx) have their own XML-based file format developed by Microsoft, which is called Office Open XML (OOXML). Its structure is shown on the figure 1.

The actual content of the document is stored in the word folder in the `document.xml` file.

LibreOffice documents also have their own XML-based format called Open Document Format (ODF) also known as OpenDocument [15]. ODF is developed by The Document Foundation. The structure of the document is shown on the figure 2.

In this case, the actual content of the document is stored in the `content.xml` file. Depicted structures may vary depending on the complexity of a document. In comparison to docx document, which has three folders files hierarchy, an odt document has a similar structure but contains additional folders such as configurations.

In the case of document generating on the basis of a template with custom keywords, the keywords might be split by office software into different tags. Therefore, this approach needs additional validation and handling of the keywords parts to merge them together.

To inquire the issue let us look at a simple document that contains the following text:

```
1
${KeyWord}${KeyWord2} and ${KeyWord3}
${KeyWord4} some text.
```

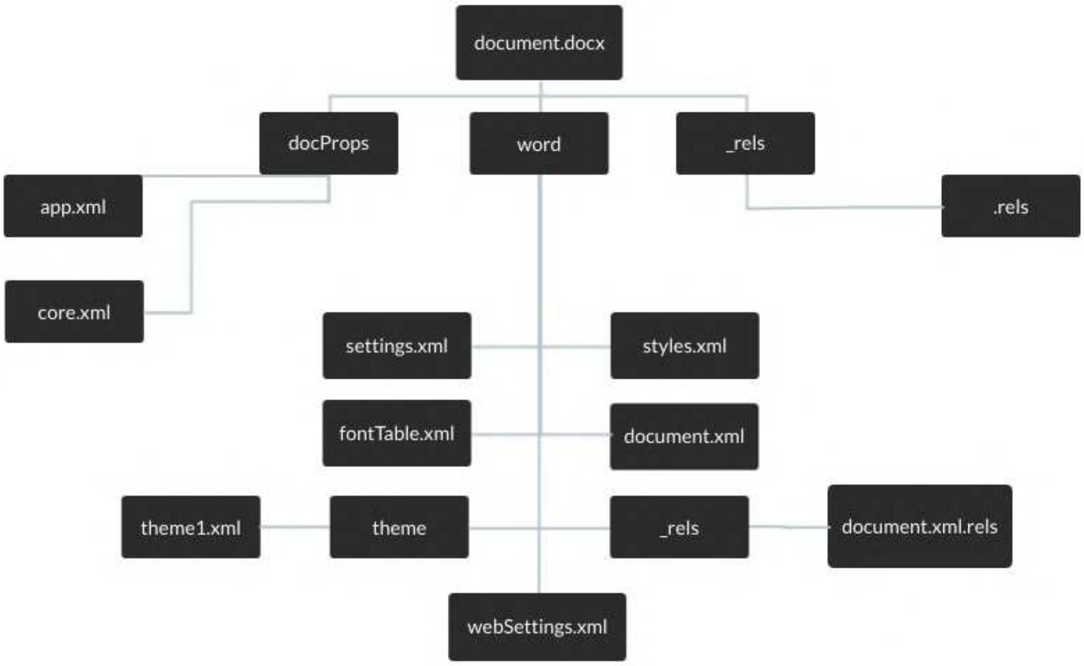


**Figure 1:** Docx file structure.

`${` and `}` statements indicate the beginning and the ending of a keyword. All the paragraphs have the same style family, namely Calibri 11 pt. However, things appear to be more surprising in the content file. Figure 3 shows the XML representation of the first keyword.

Microsoft Word splits text into different `w:r` elements called *runs*. Inside each run, we can see a `w:r` tag that represents a text element. So one keyword in this example has 3 different runs with different parts of the keyword. The second keyword is shown on figure 4.

In this case, we have four different runs. The number of runs depends on the length of the keyword and different special symbols. The same issue may be found in LibreOffice documents.

For the LibreOffice document, we will use the same font family and font size. Right after document creation, we get the solid not split paragraphs. The XML representation of the text is shown on the figure 5.

A problem may appear after editing the document with LibreOffice editor. Let's change the `KeyWord2` keyword to `KeyWord_New`. The result of this replacement is depicted on the figure 6.

As a result of a slight document editing, the XML changed significantly. New elements were added and the keyword split into 2 parts, even though the keyword still has the same style. At first glance, it may seem that the problem is in using the underscore character. However, to dispel this assumption, we will return the original value to the keyword. The result is shown on the figure 7.

Even after original value recovery, we still have the XML code which is different from the initial one. Moreover, two extra `text:span` elements appeared. In the case of the LibreOffice

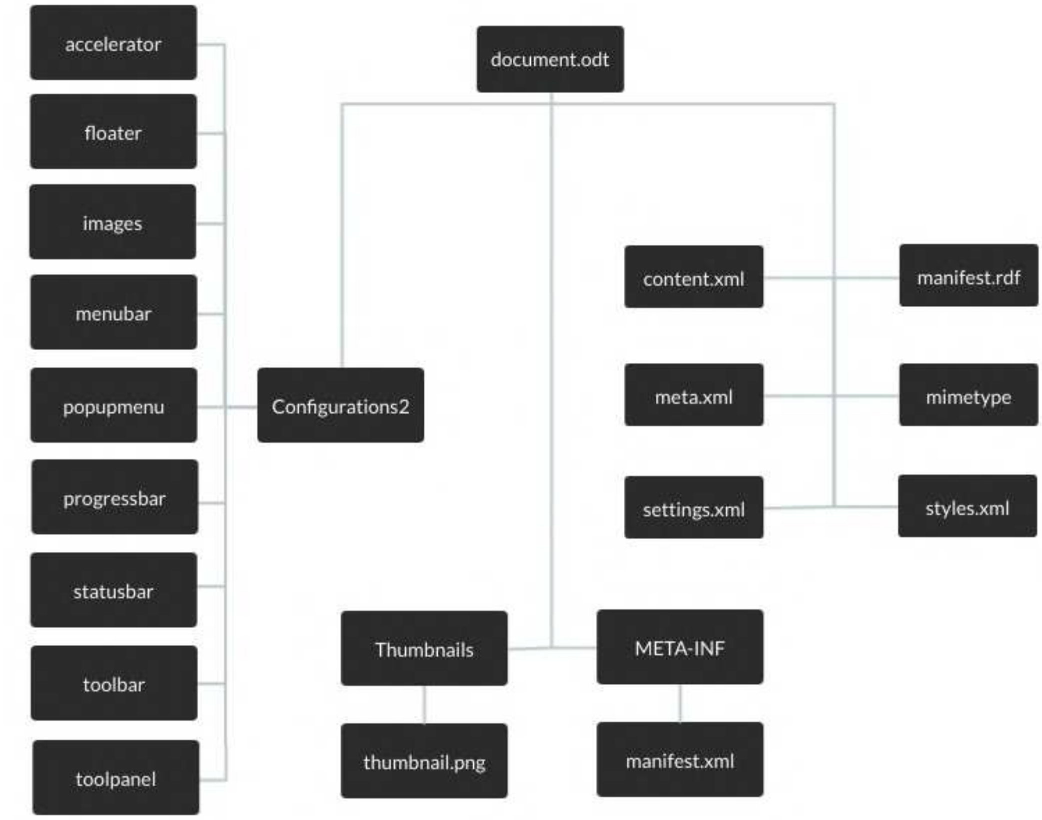


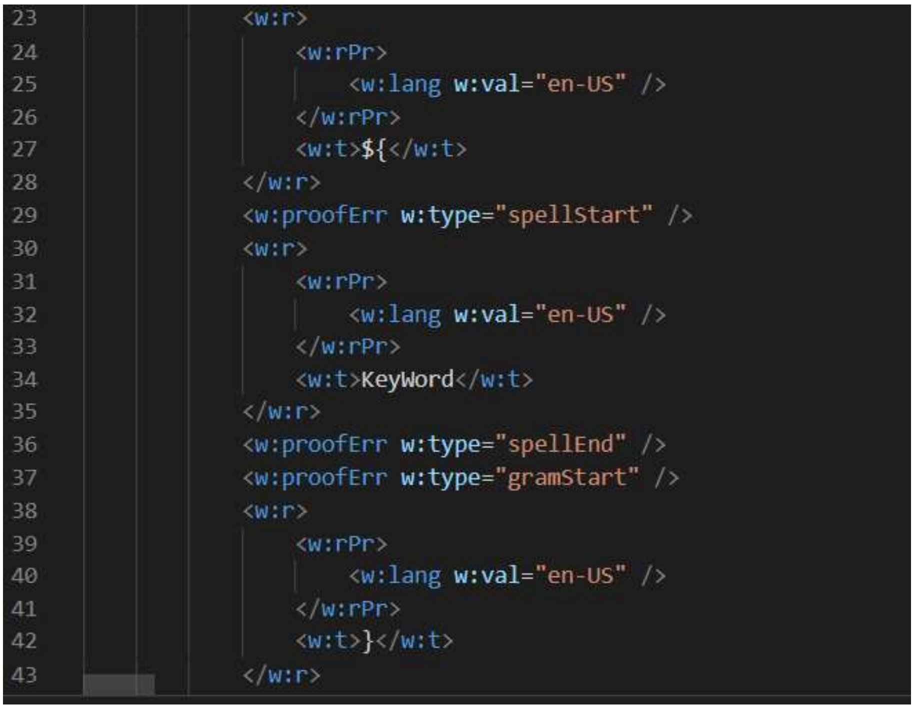
**Figure 2:** ODT file structure.

documents, `text:span` elements may be added as a consequence of updating or text changing within the document.

Another approach is using LibreOffice UNO API. LibreOffice provides Universal Network Objects, which allows using this API in different programming languages, such as C++, Java, Python, Perl, C#, JavaScript, and many others. This API supports working with different formats, originally LibreOffice applications, but partly including support of Microsoft Office applications. As a matter of fact, LibreOffice UNO API is almost completely compatible with OpenOffice.

LibreOffice has a Frame-Controller-Model paradigm (FCM) that is similar to the Model-View-Controller paradigm (MVC) [16]. The model contains the document data and methods to change them. The controller views the status of the documents and manipulates screen presentations. The frame contains the controller and knows which windows are being used. This approach allows interacting easily with the application's GUI and its functionality.

LibreOffice UNO API is extremely functional and useful in document manipulation. However, API documentation is bulky and might be time-consuming to read [17]. Due to this fact, we decided to develop a library as a layer over the LibreOffice UNO API.



```
<w:r>
    <w:rPr>
        <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>${</w:t>
</w:r>
<w:proofErr w:type="spellStart" />
<w:r>
    <w:rPr>
        <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>KeyWord</w:t>
</w:r>
<w:proofErr w:type="spellEnd" />
<w:proofErr w:type="gramStart" />
<w:r>
    <w:rPr>
        <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>}</w:t>
</w:r>
```

**Figure 3:** XML representation of the first keyword.

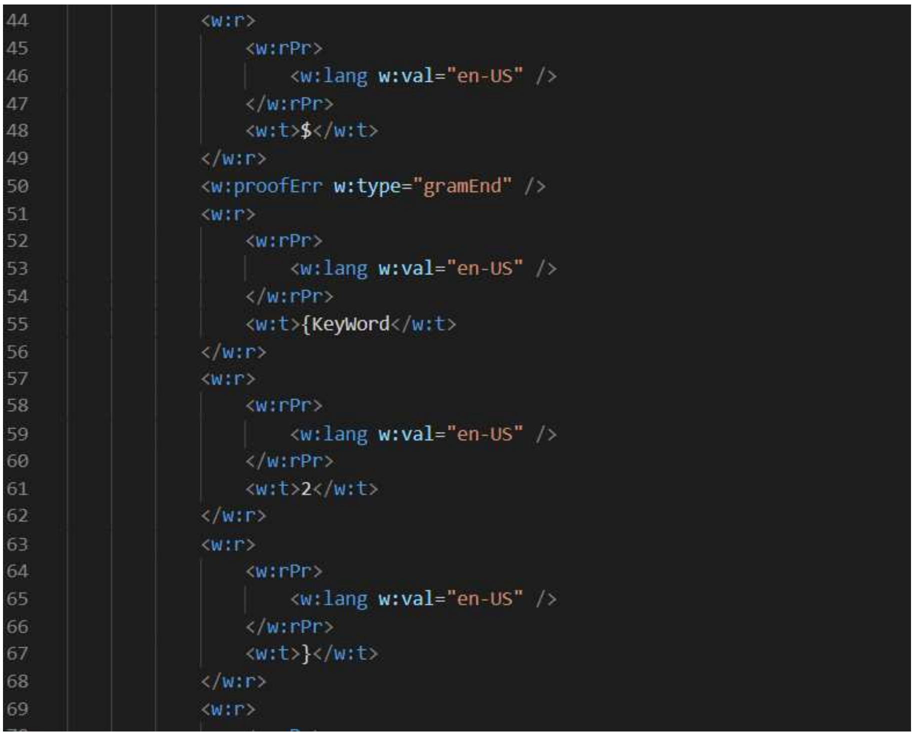
Returning to the split issue in XML documents, LibreOffice UNO API allows one to use GUI and work with text in a simpler manner. It handles text as though it had been edited by user. In addition, in comparison with the XML approach, this API provides access to styles and other functionality, like pictures, converters and other GUI functions.

We have chosen the Java programming language to work with the LibreOffice UNO API. Our library provides an abstraction to process documents easier in comparison with UNO API, and it does not require knowledge of the LibreOffice UNO API. As a part of this library, we have implemented classes for XML manipulations. In more detail, this library will be discussed in the next section.

## 4. Documents processing Java library implementation

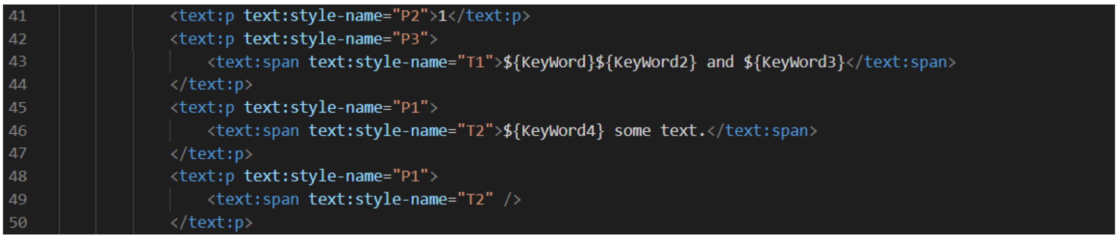
The easier document processing approach is XML processing. It allows developers to implement a simple keywords replacement. On the other hand, LibreOffice UNO API provides a rich set of functionality for document manipulation. Nevertheless, it does not nullify the usefulness of the XML approach. A combination of two different approaches allows choosing developers which one is the most appropriate for their application. Usually, one ought to use two different libraries or frameworks to implement it, but our library provides a simple interfaces to interact with both solutions simultaneously.

Our library's purpose is to simplify access to the documents and their handling by providing



```
<w:r>
    <w:rPr>
        <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>$</w:t>
</w:r>
<w:proofErr w:type="gramEnd" />
<w:r>
    <w:rPr>
        <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>{KeyWord</w:t>
</w:r>
<w:r>
    <w:rPr>
        <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>2</w:t>
</w:r>
<w:r>
    <w:rPr>
        <w:lang w:val="en-US" />
    </w:rPr>
    <w:t>}</w:t>
</w:r>
<w:r>
```

**Figure 4:** The XML representation of the second keyword.



```
<text:p text:style-name="P2">1</text:p>
<text:p text:style-name="P3">
    <text:span text:style-name="T1">${KeyWord}${KeyWord2} and ${KeyWord3}</text:span>
</text:p>
<text:p text:style-name="P1">
    <text:span text:style-name="T2">${KeyWord4} some text.</text:span>
</text:p>
<text:p text:style-name="P1">
    <text:span text:style-name="T2" />
</text:p>
```

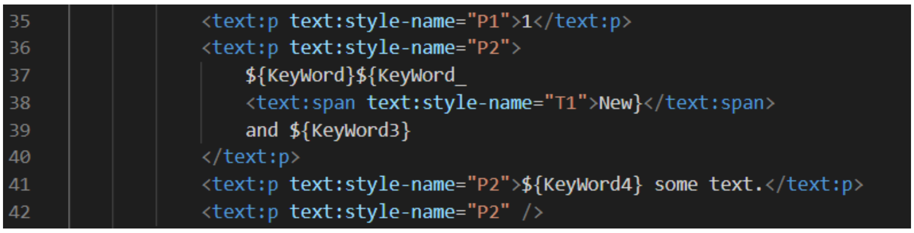
**Figure 5:** The XML representation of the text in the LibreOffice document.

an additional abstraction. The library has been implemented using the Java programming language. The source code may be found at [18].

The XML approach is quite simple to use. The main class is `OdtDocumentPatternsAdjust`. It has two constructors. The first one is empty, and the second one with a `Pattern` parameter. The `Pattern` class is a `JavaBean` class with two fields, the start of the pattern and the end of the pattern.

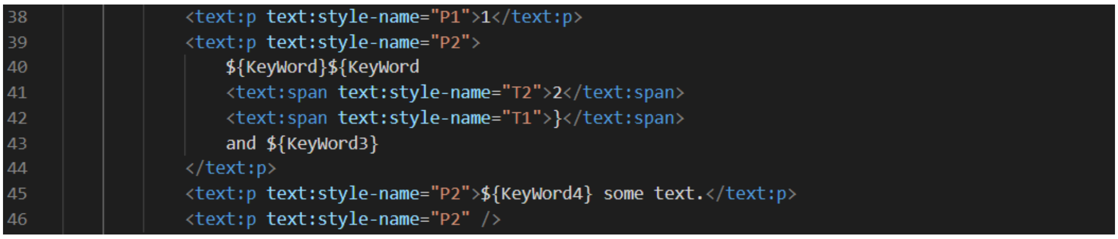
The `OdtDocumentPatternsAdjust` class implements the `DocumentPatternsAdjust` interface which has two methods for adjusting the XML content. The methods are the following:

```
String adjustPatterns(File archive)
```



```
<text:p text:style-name="P1">1</text:p>
<text:p text:style-name="P2">
    ${KeyWord}${KeyWord_
    <text:span text:style-name="T1">New}</text:span>
    and ${KeyWord3}
</text:p>
<text:p text:style-name="P2">${KeyWord4} some text.</text:p>
<text:p text:style-name="P2" />
```

**Figure 6:** The XML document after editing.



```
<text:p text:style-name="P1">1</text:p>
<text:p text:style-name="P2">
    ${KeyWord}${KeyWord
    <text:span text:style-name="T2">2</text:span>
    <text:span text:style-name="T1">}</text:span>
    and ${KeyWord3}
</text:p>
<text:p text:style-name="P2">${KeyWord4} some text.</text:p>
<text:p text:style-name="P2" />
```

**Figure 7:** The document XML after return the original value.

```
String adjustPatterns(File archive, Pattern pattern)
```

The actual processor of the XML content is the `OdtXmlPatternAdjustProcessor` class. It contains different methods for content processing, most of which are private. One of the public method is `processXml`. The algorithm of XML content processing is the following:

1. Get the position of the start and the end of the pattern.
2. Set offset to the position of the start of the pattern.
3. Get text before the next tag. It is needed to get the part of the pattern before there will be the next tag like `w:r` or `text:span`.
4. Move offset by adding the length of the found part of the pattern.
5. Look for the next possible part of the pattern meanwhile skipping tags without actual text inside.
6. When the next part of the pattern is found, get the text. At this step, the text will be extracted and inserted into the beginning of the pattern in the XML content.
7. Check whether the offset is less than the position of the end of the pattern; if it is, then repeat every action starting with step 5, otherwise the next step.
8. If the next part of the pattern could be found, repeat every action starting with step 1, and add the earlier found pattern into the `ArrayList`, otherwise return the list of the pattern.

The LibreOffice UNO API part is larger and offers richer functionality. There are a few essential classes. First of all, consider the `DocumentManagerProvider` class. This class is a `Factory` and provides the implementation of corresponding `DocumentManager` depending on file extension. This class contains one static method called `createDocumentManager` and has the following signature:

```
DocumentManager createDocumentManager(File file)
```

`DocumentManager` is an interface that provides an ability to open a passed document. It has the following method:

```
Document openDocument(File file);
```

The `openDocument` method returns a `Document` instance, which is also an interface. This approach allows avoiding specific implementations for a developer. The `Document` class contains the following set of methods:

```
void saveDocument(File file);
void saveDocument(String filepath);
void saveDocument();
void saveDocumentAs(File file, DocumentConvertTypes convertTo);
void saveDocumentAs(String filepath, DocumentConvertTypes convertTo);
void saveDocumentAs(DocumentConvertTypes convertTo);
void replace(String search, String replace);
void close();
```

These methods allow converting documents to any supported format and replacing a particular value in a document. The LibreOffice UNO API supports a considerable amount of formats to convert. All of them are described in Apache OpenOffice Wiki [19]. Partly, those types had been moved to our library and stored in an `enum` called `DocumentConvertTypes`. We decided to use enums to simplify the usage of constants that can be used as properties. In comparison with final static variables, enums make it easier to specify what should be passed there.

To provide more functionality, a lower abstraction layer is available. The `LibreOfficeUnoManager` contains most of the implemented methods in the `Document` class. This class provides basic methods to interact with documents without direct work with UNO API. As return values, it uses API's objects, so it may be considered an additional functionality layer.

There are small utility classes which might help in working with documents. Nevertheless, developers will rarely use them because most of the `LibreOfficeUnoManager` methods already have been optimized with the use of those utility classes. Let us look into two useful classes. The `OdtDocumentProperties` provides a wrapper for the `PropertyValue` class to simplify working with document properties. The `OdtFilePathHandler` helps to convert the initial File class into an understandable LibreOffice UNO API string. The reason for `OdtFilePathHandler` class existence is that LibreOffice UNO API works with Uniform Resource Identifier (URI). This means that the file path should be started with the `file:///` prefix and all backslash characters should be replaced with the slash character.

The next example demonstrates a basic usage of our library to replace keywords in the document and convert it into an appropriate format:

```
File file = ResourcesManager.getResourceFile("Document.odt");
DocumentManager documentManager =
         DocumentManagerProvider.createDocumentManager(file);
Document document = documentManager.openDocument(file);
document.replace("{Search}", "Value");
document.saveDocumentAs(new File(
    "C:/Users/hp/IdeaProjects/XmlDocumentProcessing/File.docx"),
    DocumentConvertTypes.MS_WORD_2007_XML);
```

In order to work with text, we implemented a few specific classes. The `LibreOfficeUnoManager` class supports working with text using the `findAllAsText` method. The method's signature is the following:

```
public List<Text> findAllAsText(String search);
```

This method returns a list of `Text` classes. The `Text` class supports text editing, creating cursor, getting all paragraphs, setting font weight, and paragraph adjustment. The list of the methods is shown on the Figure 8.

An example of getting a text and performing some basic operations is shown below.

```
Text allDocumentText = libreOfficeUnoManager.findAllAsText("and").get(0);
allDocumentText.createCursor().gotoStartOfTheSentence(true);
allDocumentText.setCenteredAdjustment();
```

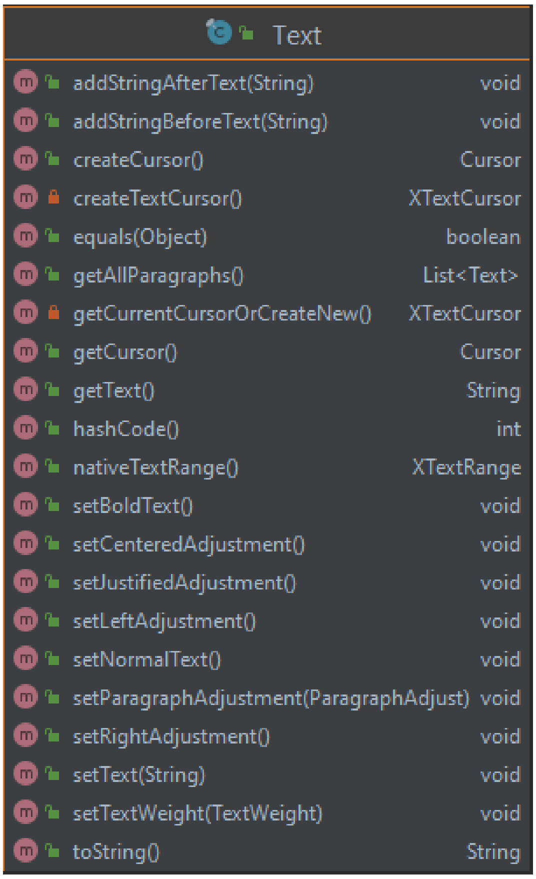
The `Cursor` class is basically usual graphic cursor. In order to move through the text, LibreOffice UNO API implements cursor as a main mechanism for this purpose. But, considering the fact of complexity of some original UNO API methods, we have implemented a simplified wrapper class. The list of its methods is shown on the figure 9.

The names of most methods, such as `gotoNextSentence`, are intuitively recognizable. Every type of `goto` moving has two different implementations. One does not have parameters and another one has a Boolean parameter. The `Boolean` parameter is used for telling the LibreOffice UNO API, whether should we stop and select current word or go to next one. Methods without parameters basically just use methods with parameters by passing `false` to them.

Also, to implement a more convenient way of `Cursor` class methods usage, `goto` methods take advantage of `Builder` design pattern. The example of such use is shown below.

```
allDocumentText.createCursor()
.gotoStartOfTheSentence()
.gotoNextSentence()
.gotoNextWord()
.gotoPreviousWord();
```

It is impossible to predict different components usage due to LibreOffice UNO API complexity and massiveness. So, to simplify it for developers, all the classes contain corresponding methods



Text

| | |
|---|---|
| addStringAfterText(String) | void |
| addStringBeforeText(String) | void |
| createCursor() | Cursor |
| createTextCursor() | XTextCursor |
| equals(Object) | boolean |
| getAllParagraphs() | List<Text> |
| getCurrentCursorOrCreateNew() | XTextCursor |
| getCursor() | Cursor |
| getText() | String |
| hashCode() | int |
| nativeTextRange() | XTextRange |
| setBoldText() | void |
| setCenteredAdjustment() | void |
| setJustifiedAdjustment() | void |
| setLeftAdjustment() | void |
| setNormalText() | void |
| setParagraphAdjustment(ParagraphAdjust) | void |
| setRightAdjustment() | void |
| setText(String) | void |
| setTextWeight(TextWeight) | void |
| toString() | String |

**Figure 8:** The list of `Text` class methods.

which return the original LibreOffice UNO API objects. For instance, the `Cursor` class has `getTextCursor`, which returns a `XTextCursor` object.

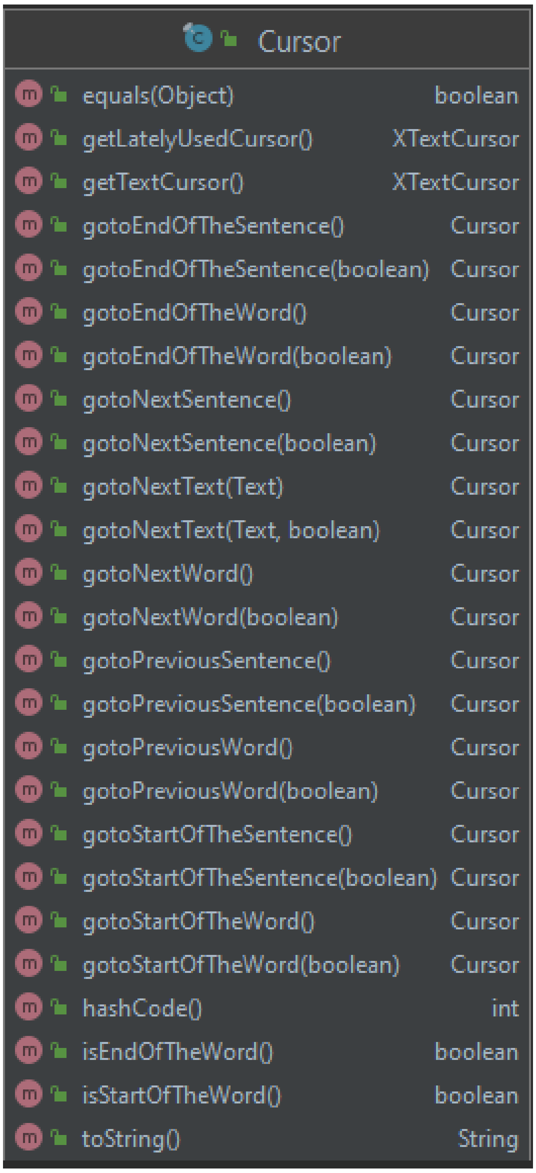
In order to demonstrate the developed library usage, we implemented a cloud-based system which aims to automate document flow.

The application is divided onto frontend and backend parts. The development stack is shown below:

- Server development stack: Spring (Spring boot, Spring Security, Spring WebFlux, Spring JPA), jjwt (Java JWT: JSON Web Token for Java and Android), Connector/J (Mysql Java Connector).
- Client development stack: Vue.js 3 (Vue Cli, Vue Router, Vuex, Vue i18n, Vue Class Component, Vue FontAwesome, SFC, Element Plus), Typescript, Javascript, Babel, Webpack.

The backend has microservice architecture. In order to minimize the application load, we have implemented 3 different microservices:

1. Microservice for login and token generation.



Cursor

| | |
|---|---|
| equals(Object) | boolean |
| getLatelyUsedCursor() | XTextCursor |
| getTextCursor() | XTextCursor |
| gotoEndOfTheSentence() | Cursor |
| gotoEndOfTheSentence(boolean) | Cursor |
| gotoEndOfTheWord() | Cursor |
| gotoEndOfTheWord(boolean) | Cursor |
| gotoNextSentence() | Cursor |
| gotoNextSentence(boolean) | Cursor |
| gotoNextText(Text) | Cursor |
| gotoNextText(Text, boolean) | Cursor |
| gotoNextWord() | Cursor |
| gotoNextWord(boolean) | Cursor |
| gotoPreviousSentence() | Cursor |
| gotoPreviousSentence(boolean) | Cursor |
| gotoPreviousWord() | Cursor |
| gotoPreviousWord(boolean) | Cursor |
| gotoStartOfTheSentence() | Cursor |
| gotoStartOfTheSentence(boolean) | Cursor |
| gotoStartOfTheWord() | Cursor |
| gotoStartOfTheWord(boolean) | Cursor |
| hashCode() | int |
| isEndOfTheWord() | boolean |
| isStartOfTheWord() | boolean |
| toString() | String |

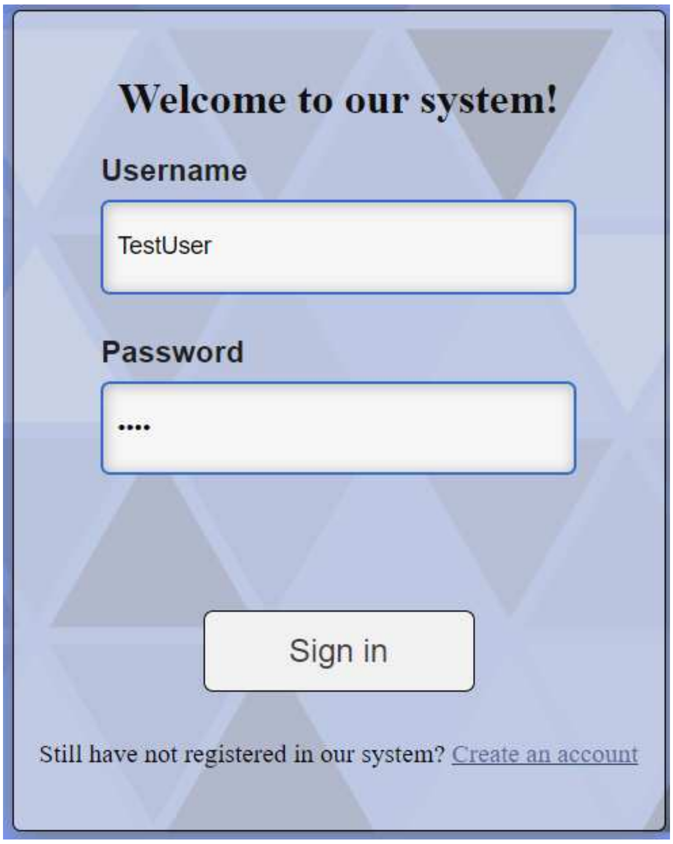
**Figure 9:** The list of `Cursor` class methods.

2. Microservice for document processing (storage and document management).
3. Microservice for generating documents according to the data.

As a matter of application security and microservice communication, we have used the JWT token as the most eligible.

For signing up and signing in into the application, the login page may be used (figure 10).

After this procedure, the user goes to the main page for handling documents, which is called Document Management (figure 11).



Welcome to our system!

Username

TestUser

Password

••••

Still have not registered in our system? Create an account

**Figure 10:** The login page.

This page contains all the document information. To create a document, one should push a side bar button which leads to a document adding page (figure 12). The index of the documents is shown as a list, and each item has two different buttons:

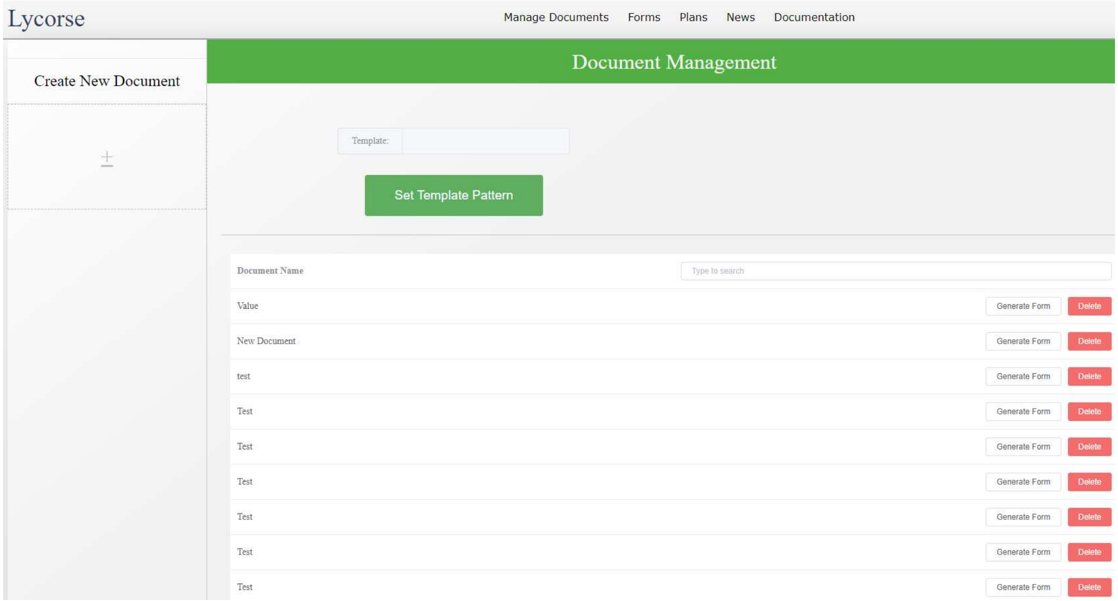
1. Generate Form. This button is responsible for form generating. These forms may be used as data origins for producing documents from templates.
2. Delete. Remove the entry.

Furthermore, our system supports custom template patterns, which means that documents may contain any kind of keyword distinguishers.

The forms are common way of document generating from an uploaded template. All created forms are displayed and might be changed in the Form page (figure 13).

The actual form page, which may be accessed by using the View Form button, contains all of the extracted from the template document keywords. The example of a form is shown on the figure 14.

Considering the fact that key words are not always named human-friendly, it is also possible to change their display name using the Edit button on the table. After submitting a form, the user automatically receives the document.



| Document Name | | |
|---|---|---|
| Value | Generate Form | Delete |
| New Document | Generate Form | Delete |
| test | Generate Form | Delete |
| Test | Generate Form | Delete |
| Test | Generate Form | Delete |
| Test | Generate Form | Delete |
| Test | Generate Form | Delete |
| Test | Generate Form | Delete |
| Test | Generate Form | Delete |

**Figure 11:** The Document Management page.

At the moment, the following features have been implemented:

1. The XML adjuster.
2. Document handling without working with UNO API directly.
3. Simplified classes for working with UNO API and its components.
4. Utilities for working with URI.
5. Converters.
6. Constants classes implemented as an enum class.
7. A cloud-based system for documents automation.
8. Interfaces and classes which help to add custom implementations of most mechanisms of the library.

We are planning to implement a cloud-based interface for working with documents without coding. In addition, we have intention to provide the richest functionality for working with LibreOffice UNO API.

## 5. Conclusion

Document processing may be complicated and confusing. The XML processing is more complicated and limited. The reason is that handling raw XML is difficult, especially when a document is massive.

LibreOffice UNO API is one of the richest open-source APIs for processing documents. It provides the necessary functionality to edit and process documents. In comparison with XML

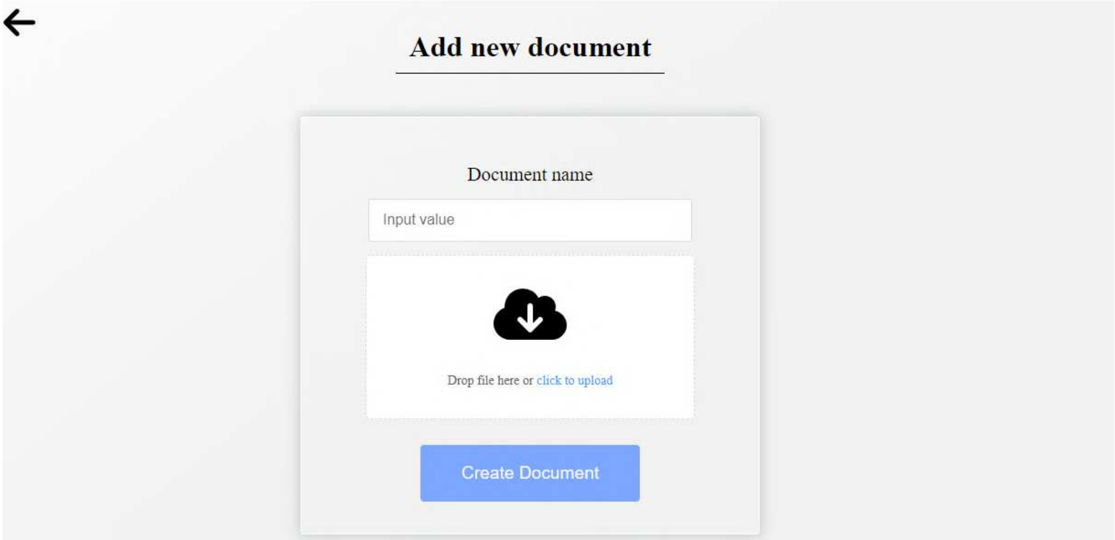
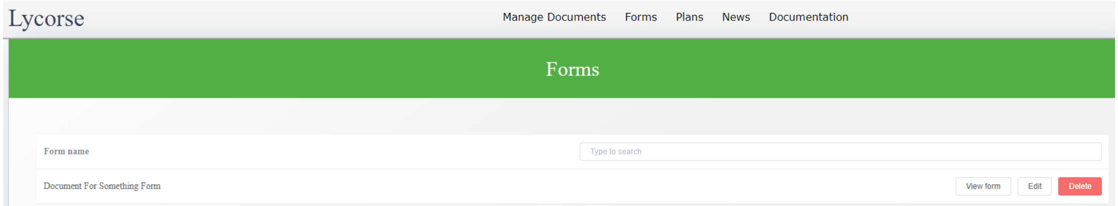


**Figure 12:** The document adding page.



| Form name |
|---|
| Document For Something Form |

**Figure 13:** The Form page.

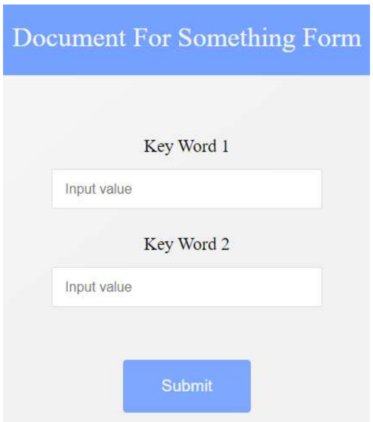


**Figure 14:** The generated form page.

processing, this approach is more advantageous. Moreover, the LibreOffice UNO API solves the keyword splitting issue, or to be more precise, allows avoiding it.

The developed library allows one to handle both of the described processing approaches. It is easier to combine them regardless of whether you only need to process patterns or additionally edit the inner structure of the document. Looking at the future, we are planning to complete the development of this library. Converters of the library are useful tools because there are not many solutions that could manage all the major formats, such as doc, docx, odt, html, and others. As an application of this library, we are currently working on creating a cloud-based document management system that will be able to help in storing, handling, and processing documents. It is going to be discussed in the further reports.

## References

[1] IBM Corporation, The evolution of process automation, 2018. URL: https://www.ibm.com/downloads/cas/QAQMRGVN.

[2] McKinsey&Company, Jobs lost, jobs gained: Workforce transitions in a time of automation, 2017. URL: https://www.mckinsey.com/~/media/BAB489A30B724BECB5DEDC41E9BB9FAC.ashx.

[3] S. Wilson, How document automation is changing the healthcare industry, 2017. URL: https://electronichealthreporter.com/document-automation-changing-healthcare-industry/.

[4] M. Bhanja, N. Barik, Library automation: problems and prospect, in: 10th National Convention of MANLIBNET organized by KIIT University, 2009, pp. 199–201. URL: https://www.researchgate.net/publication/323219596_Library_Automation_problems_and_prospect.

[5] H.-Y. Hsueh, C.-N. Chen, K.-F. Huang, Generating metadata from web documents: a systematic approach, Human-centric Computing and Information Sciences 3 (2013). doi:10.1186/2192-1962-3-7.

[6] S. T. Rosenbloom, W. Kiepek, J. Belletti, P. Adams, K. Shuxteau, K. B. Johnson, P. L. Elkin, E. K. Shultz, Generating complex clinical documents using structured entry and reporting, Studies in health technology and informatics 107 (2004) 683–687. URL: https://pubmed.ncbi.nlm.nih.gov/15360900/.

[7] M. J. A. Salomi, R. F. Maciel, Document management and process automation in a paperless healthcare institution, Technology and Investment 08 (2017) 167–178. doi:10.4236/ti.2017.83015.

[8] Hypatos, Hypatos document hyperautomation for e2e doc processing, 2021. URL: https://hypatos.ai/en.

[9] C. Dilmegani, The ultimate guide to document automation in 2021, 2021. URL: https://research.aimultiple.com/document-automation/.

[10] Docuphase, Enterprise automation software, 2021. URL: https://www.docuphase.com/.

[11] Flackon Inc., Document automation software, 2021. URL: https://docupilot.app/.

[12] Contractbook, Better contracts, 2021. URL: https://contractbook.com/.

[13] Docassemble, Docassemble, 2021. URL: https://docassemble.org/.

[14] Obeo, M2doc, 2021. URL: https://www.m2doc.org/.

[15] Wikipedia, Opendocument, 2021. URL: https://en.wikipedia.org/w/index.php?title=OpenDocument&oldid=1025760709.

[16] Apache, Frame-controller-model paradigm in apache openoffice, 2021. URL: https://wiki.openoffice.org/wiki/Documentation/DevGuide/OfficeDev/Frame-Controller-Model_Paradigm_in_OpenOffice.org.

[17] A. Davison, Java libreoffice programming, 2021. URL: https://fivedots.coe.psu.ac.th/~ad/jlop/.

[18] CodePsi, GitHub - CodePsi/Lycorse-DPL: Lycorse Document Processing Library, 2021. URL: https://github.com/CodePsi/Lycorse-DPL.

[19] Apache, Framework/article/filter/filterlist ooo 3 0, 2021. URL: https://wiki.openoffice.org/wiki/Framework/Article/Filter/FilterList_OOo_3_0.